Exhibit A

Cyberdog ERS

(Preliminary)

Executive Summary

At a high level, the Cyberdog project is a blending of old and new technologies. The old technology — existing Internet services and content viewers — are in themselves reasonably straightforward. It is the new technology that presents the real challenge: making these services and viewers work within OpenDoc, and creating a unifying environment in which they can interoperate smoothly. The engineering requirements for the Cyberdog project thus fall into four major categories:

Developing a Public API

From the initial concept, Cyberdog has been envisioned as an extensible platform for developing Internet (and other network) services. The technical challenge is to create an API which is flexible enough to accommodate a wide variety of Internet services and content viewers, some of which have not yet been envisioned. Fortunately, the OpenDoc architecture addresses many of the same issues, and thus most of Cyberdog's public API is in fact the OpenDoc API. The Cyberdog API consists of a subclass of XMPPart and adds three other abstract classes to accommodate the specific needs of working with heterogeneous data and protocols.

Developing an Internal Framework

The OpenDoc Technical Summary claims "the purpose of OpenDoc's design is not to make it as *simple as possible* to create compound documents, but merely to make it *possible*." This is an accurate assessment. To efficiently engineer OpenDoc parts requires more infrastructure than the raw OpenDoc API provides. The OpenDoc team is developing a framework for this, but it is in the early stages of development, and will not be stable early enough for our needs. We are therefore developing our own framework (inspired and influenced by that of the OpenDoc team) that meets our specific needs. The most complex aspect of this framework is handling multiple views onto the same data (frames and facets in OpenDoc parlance). The Cyberdog framework is largely designed around this issue, enforcing a clean separation of data and presentation and providing mechanisms for keeping everything in synch.

Implementing Cyberdog Core Functionality

The two previous sections describe architectural challenges, but of course the product must be implemented as well. The novel part of the implementation involves building the unifying elements of Cyberdog, the log and the favorite places book, and making them work with both OpenDoc and our own extensible architecture. To make this problem tractable, the Cyberdog implementation relies on the abstractions of the public API to allow us to treat heterogeneous objects uniformly, and our internal framework to support the requirements of OpenDoc

Implementing Internet Services

This is the simplest issue facing the Cyberdog team, since existing Internet protocols are stable and already have numerous implementations. In general, this category represents a lot of work but it does not require much invention. (A notable exception is NetNews, an efficient implementation of which requires considerable design.)

Public API

Goals

Because new services and content appear frequently on the Internet, Cyberdog needs to be easily extensible. Specifically, Cyberdog is designed to:

- Enable third parties to write new Cyberdog services or to replace existing Cyberdog services. To the user, these services should appear completely integrated with the Cyberdog user experience. In particular, these services should be able to take full advantage of the favorite places book and the automatic log.
- Allow the Cyberdog team and third parties to write a set of content viewers that can display various media formats without resorting to helper applications.
- Enable these content viewers to display their content regardless of the source of the underlying data: HFS, GopherSpace, the World Wide Web, etc.
- Allow users to create and manipulate small, portable "aliases" to their favorite locations on the Internet regardless of the tools or protocols necessary to reach that location.
- Smoothly integrate with OpenDoc, allowing documents to embed parts that provide a window onto the Internet.

Core Classes

Because Cyberdog is so extensible, its core framework makes no assumptions about protocols or data formats. Instead, the framework relies on a set of wrapper classes that provide a common API for all services and abstract away the particulars of each protocol.

The section below describes the core classes in the public Cyberdog API. These classes are abstract base classes; both the Cyberdog engineering team and third party developers must create subclasses as appropriate.

CyberItem

- CyberItems are small, self-contained object representing a single resource on the Internet. Subclasses can represent Gopher directories, Mosaic pages, Newsgroups, etc.
- CyberItems have methods for returning an icon family and a name. This allows the log and user pages in the favorite places book to display CyberItem.
- CyberItems know how to create the appropriate subclass of CyberPart (see below) to display the data they represent.
- Each CyberItem stores information on how it wishes to be displayed. This may include font information, frame sizes, etc.
- Some CyberItems may need to know how to create a CyberStream (see below) to obtain the actual data for the object they represent.
- When initialized to an empty state, CyberItems know how to prompt the user for whatever information is necessary to create a complete CyberItem. For example, an empty Gopher CyberItem will know how to prompt the user for a host name, path, and port.

CyberPart

- OpenDoc part responsible for displaying a single CyberItem.
- CyberParts are classified as either browsers or viewers. Note that the difference between browsers and viewers is strictly semantic — they both use the same CyberPart API.
- A browser CyberPart will display objects that represent additional CyberItems (e.g., other Gopher directories, newsgroups, or URLs that are contained in the current CyberItem), and it knows how to create CyberItems for these objects.
- A viewer CyberPart displays only media content such as pictures or text. A viewer may need to ask its CyberItem for a CyberStream to obtain data over the network.
- Subclasses of CyberPart inherit behavior that allows them to access the favorite places book and the automatic log.

CyberStream	• A CyberStream is an abstraction that serves as an API between a viewer CyberPart and the actual data for that part. This allows developers to write viewers that can display content regardless of the protocol required to obtain the data. For example, a developer could write a picture viewer that uses the CyberStream API to obtain the bytes that describe a pictur e. The bytes themselves will be obtained by a subclass of CyberStream that speaks the Gopher or HTTP protocols.
	• CyberStreams are created by CyberItems. These streams are used by CyberParts to obtain the data they need to display.
CyberService	• Each CyberService knows how to display and allow the user to modify its preferences on the preferences page in the favorite places book.
	• CyberServices may contain an engine for performing asynchronous tasks. For example, a NetNews CyberService will need an engine to check for unread news.
	• All engines belonging to CyberServices are initialized when the favorite places book is opened. If there is no book, they can be created when needed.

Programmer Scenario: Writing a Picture Viewer

Conceptually, writing a picture viewer for Cyberdog is quite simple. We do not need to subclass CyberItem — there will be GopherItems, MosaicItems or other CyberItem subclasses that represent the location of pictures. There is also no need to subclass CyberStream, since there will be GopherStreams, MosaicStreams, etc. to obtain the actual bytes that make up the picture.

The only work that needs to be done is to create a subclass of CyberPart that:

1) Stores the CyberItem it is supposed to display
2) Asks the CyberItem for a CyberStream to obtain the data
3) Displays the picture on the screen

Since CyberItems and CyberStreams are abstract superclasses, a picture viewer written in this manner can display a picture regardless of its location on the Internet or the protocol required to download the data.

Programmer Scenario: Writing a Telnet Service

As a simple example of the steps a developer needs to take to add new Internet functionality to Cyberdog, we will consider adding support for Telnet.

Step 1: Subclass CyberItem

We will create a subclass of CyberItem called TelnetItem that adds fields for storing information about the host this item represents as well as the user's terminal type, backspace preferences, etc. A TelnetItem needs to be able to prompt the user for this information in the case that it is empty.

Step 2: Subclass CyberPart

We will create a subclass of CyberPart called TelnetViewer that will act as a terminal emulator and display the contents of the Telnet session in a window. The TelnetViewer can communicate directly with the appropriate host using the Telnet protocol, so it is not necessary to use a CyberStream. Contrast this with the picture viewer discussed above.

Step 3: Subclass CyberService

There are certain default preferences for the Telnet service which need to be set up on the preferences of the favorite places book. A subclass of CyberService called TelnetService will have methods for allowing the user to modify these preferences.

Programmer Scenario: Writing a Gopher Browser

The steps required to write a Cyberdog Gopher browser are:

Step 1: Subclass CyberItem

We will create a subclass of CyberItem called GopherItem. Since GopherItems need to represent text, pictures, search documents, or gopher directories, each GopherItem will have some internal state that describes the specific type of object it represents. Each GopherItem will also store the data necessary to reach the object: host, port, and selector string as well as any display preferences set by the user.

GopherItems know which CyberPart is appropriate to display their content. For example, GopherItems that represent pictures know that they must invoke the PictureViewer to display themselves, while GopherItems that represent Gopher directories will invoke the GopherBrowser described below. Each GopherItem also knows the correct icon it should use when displayed in a list or tree view.

Finally, an empty GopherItem needs to be able to prompt the user for a host, port, and selector.

Step 2: Subclass CyberStream

Because a PictureViewer will be displaying a GopherItem that represents a picture, the PictureViewer needs to be able to get the data for that picture without knowing the details of the Gopher protocol. To accomplish this, we must write a subclass of CyberStream called GopherStream.

Step 3: Subclass CyberPart

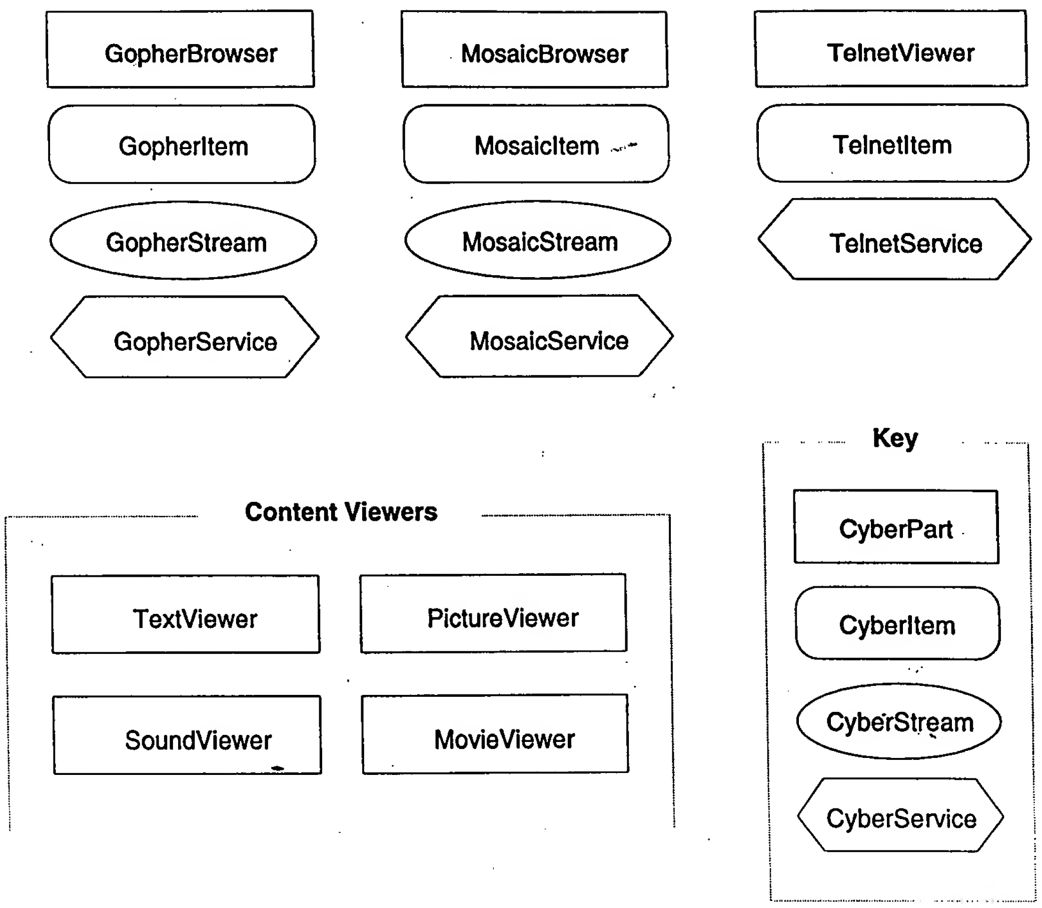
A subclass of CyberPart called GopherBrowser will know how to display a GopherItem representing a Gopher directory. (Other types of data represented by GopherItems — text, pictures, sounds, etc. — will be displayed by viewers.) Since a GopherBrowser displays objects representing other GopherItems, it needs to be able to actually create GopherItems for these objects when, for example, the user double-clicks or drags.

Step 4: Subclass CyberService

Finally, we need to create a subclass of CyberService called GopherService that handles the default preferences for GopherBrowsers.

Core Classes Summary

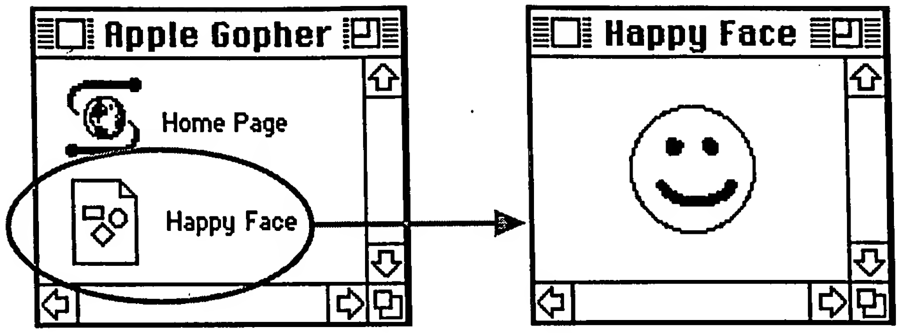
The illustration below shows a conceptual overview of the Cyberdog class structure. (Not all classes are represented.) Developers wishing to add a new browser must create subclasses of CyberPart, CyberItem, CyberStream, and CyberService (or a subset of these classes). Developers wishing to create a content viewer need only subclass CyberPart.



A subset of the core subclasses in Cyberdog

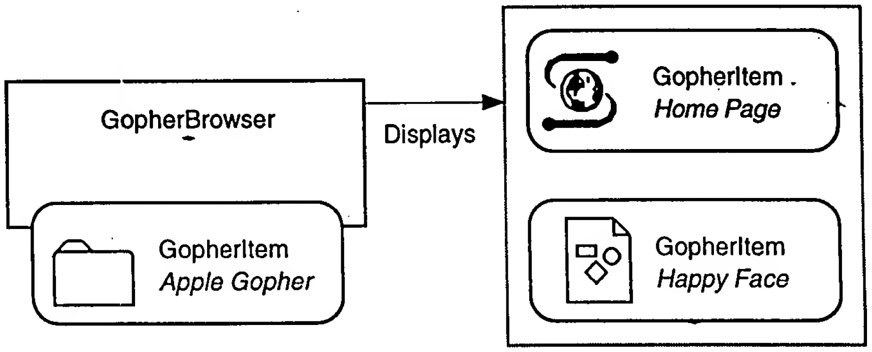
User Scenario: Displaying a Picture from Gopher

The next three sections demonstrate the interactions among CyberParts, CyberItems, and CyberStreams. As a first example, we will trace the calls necessary to display a picture from a Gopher Browser. The window on the left shows the contents of the Apple Gopher directory which consists of a Mosaic page and a picture. In this scenario, the user double-clicks on the icon for the picture.



The user double clicks on the Happy Face

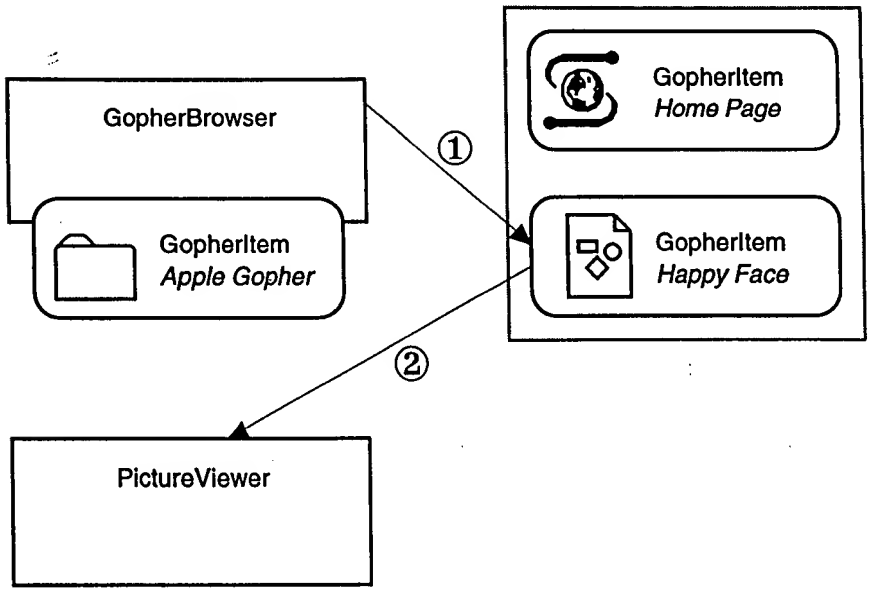
The next picture shows the internal representation of the scene shown above. There is a GopherBrowser (subclass of CyberPart) which is displaying the contents of the CyberItem it is responsible for. In this case that CyberItem is the "Apple Gopher" directory. The browser displays two additional GopherItems, one representing a Mosaic page and the other representing a picture.



Internal representation of the previous picture

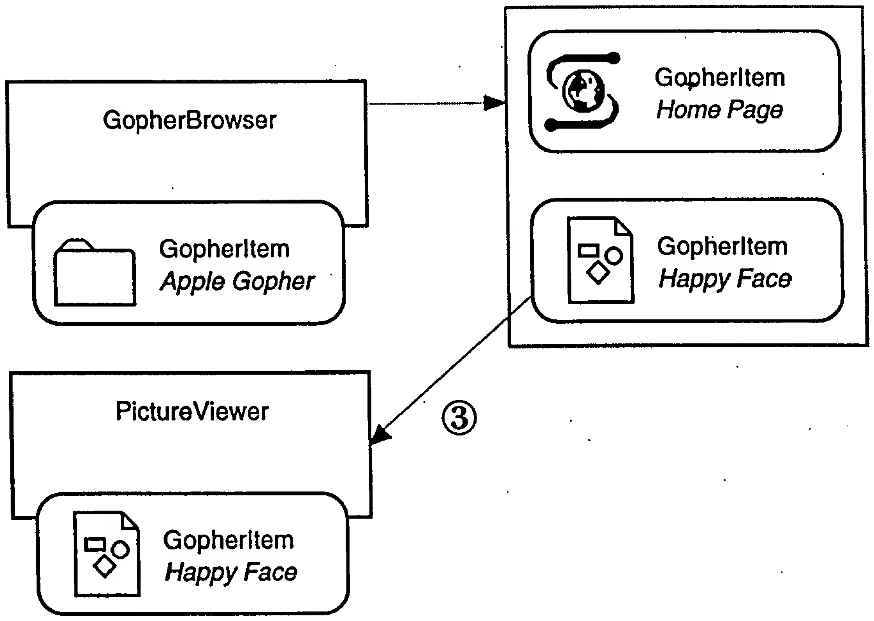
① When the GopherBrowser detects a double-click on the object representing the "Happy Face" item, it sends an Open message to "Happy Face." (At this point, the GopherBrowser can also add "Happy Face" to the automatic log.)

② The Open method of the "Happy Face" item first creates a CyberPart of the appropriate type, which, in this case, is a PictureViewer.



The "Happy Face" GopherItem creates a PictureViewer to display itself.

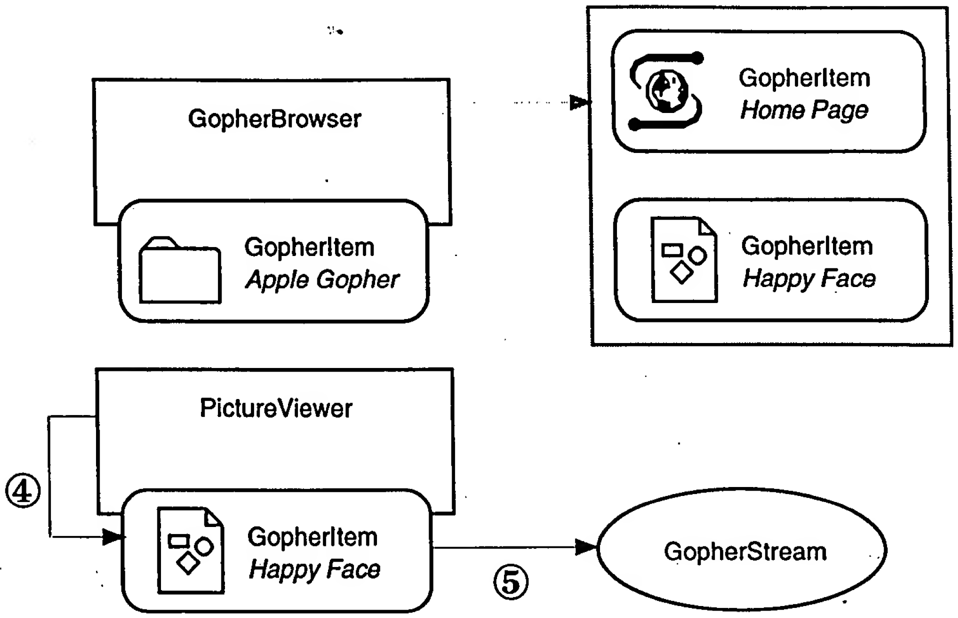
③ Next, the "Happy Face" item's Open method sets the PictureViewer's CyberItem to be a copy of itself by calling SetCyberItem. After this call returns (which occurs after step 5), it sends the PictureViewer a standard OpenDoc Open message so that it will display itself.



The "Happy Face" GopherItem assigns a CyberItem to the PictureViewer

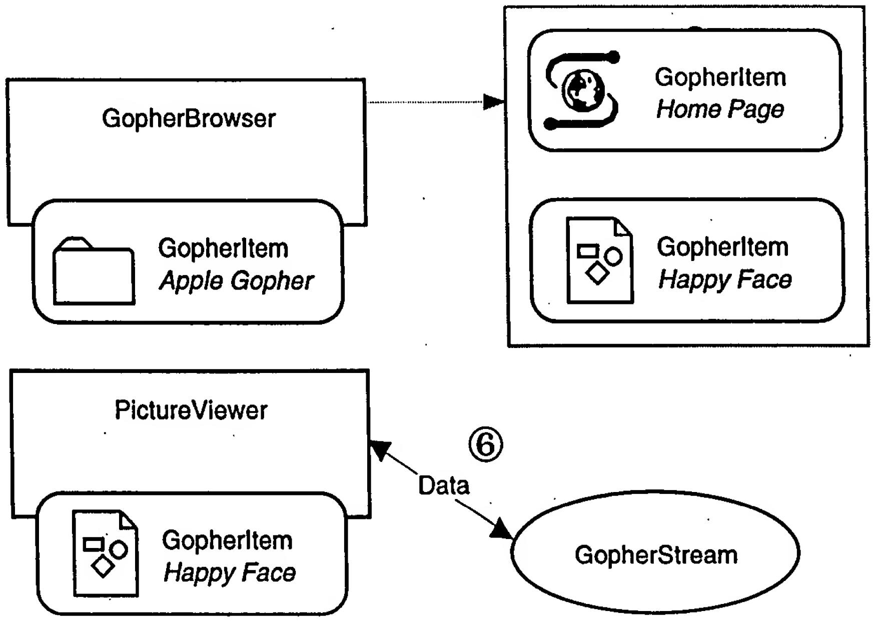
④ First, however, the PictureViewer's SetCyberItem method asks the GopherItem for a CyberStream by calling its CreateCyberStream method.

⑤ This method creates a stream that is initialized to connect to the appropriate Gopher server.



The GopherItem creates a GopherStream on behalf of the PictureViewer

⑥ Finally, the picture viewer can download the data from the Gopher server using the CyberStream API. As the data is downloaded, it can display the picture in its window.



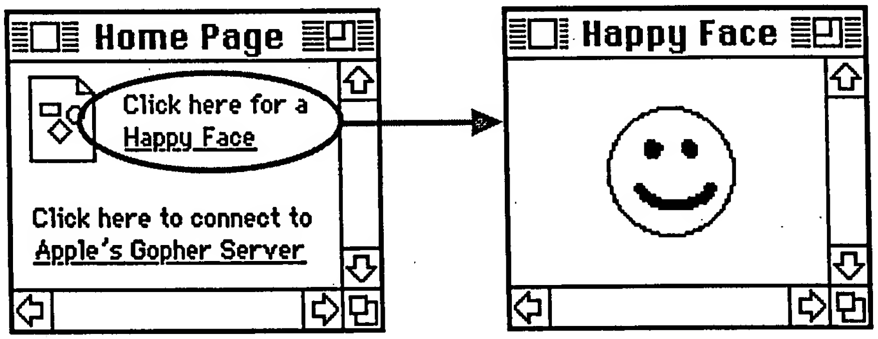
The PictureViewer uses the stream to download and display the picture

Here is a summary of the calls shown above:

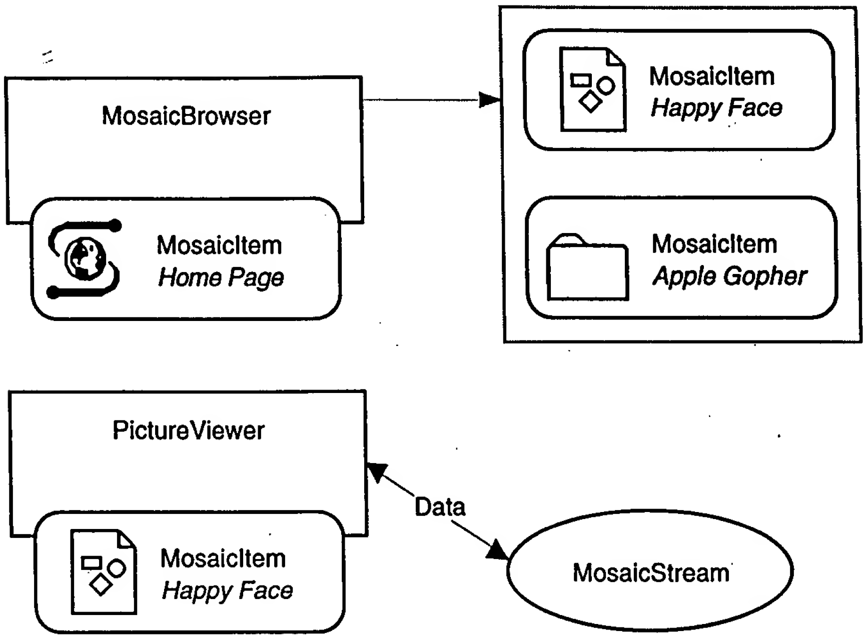
```
        GopherBrowser::DoCommand
①            CyberItem::Open
②                XMPDraft::CreatePart(kPictureViewerKind)
③                CyberPart::SetCyberItem
                     CyberItem::Clone
④                    CyberItem::CreateCyberStream
⑤                        new TGopherStream
             CyberPart::AddCyberItemToLog
...
        PictureViewer::DoIdle
⑥            CyberStream::GetData
```

User Scenario: Displaying a Picture from Mosaic

This scenario shows the data relationships resulting from the user clicking on a Mosaic item which represents a picture. The sequence of calls is identical (and therefore omitted) with only the particular subclasses differing. The key point is that the same PictureViewer can display a picture from Mosaic or from Gopher, or from anywhere else. The CyberStream subclasses (MosaicStream, GopherStream, etc.) hide the details of the connection from the viewer.


Home Page
Click here for a Happy Face
Click here to connect to Apple's Gopher Server
Happy Face

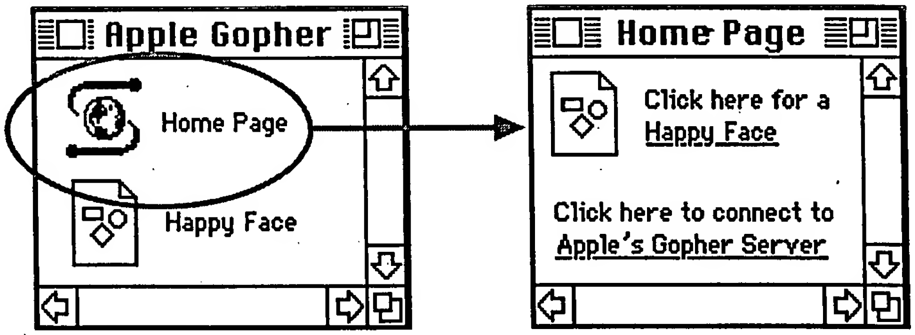
The user clicks on a Mosaic item representing a picture



Data relationships resulting from opening a picture from Mosaic

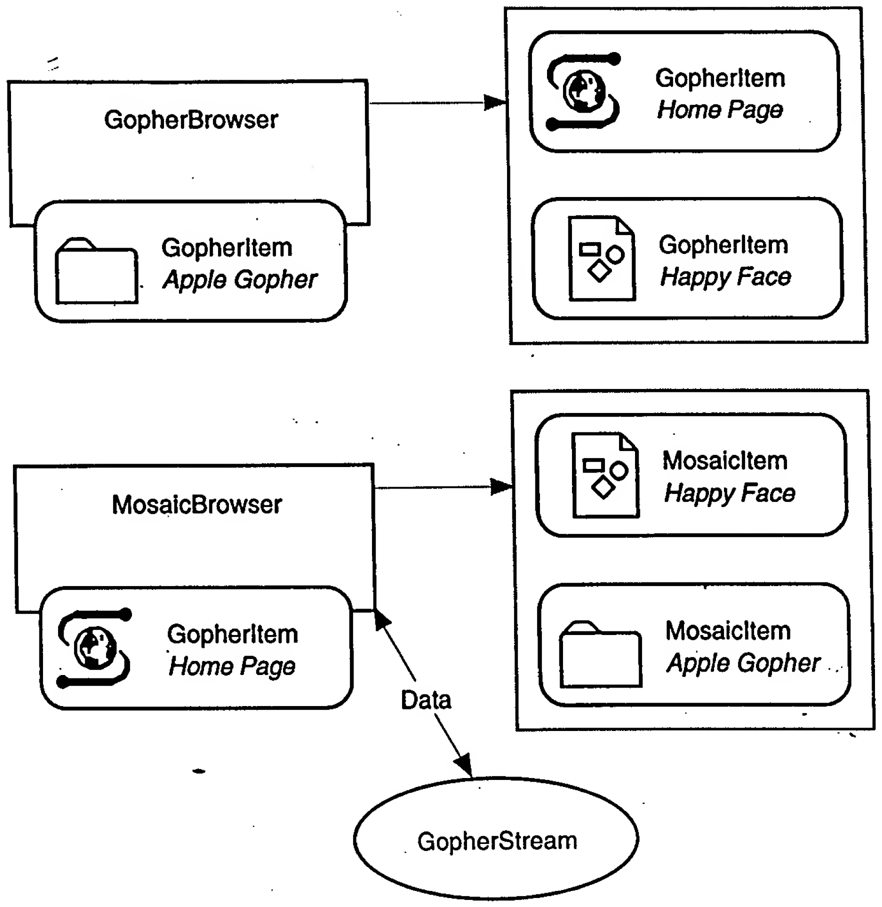
User Scenario: Displaying a Mosaic Page from Gopher

The final scenario demonstrates how the Cyberdog architecture can integrate what, has until now, been separate domains. In this scenario, a user double-clicks on an icon representing a Mosaic page from within a Gopher browser.


Apple Gopher
Home Page
Happy Face
Home Page
Click here for a Happy Face
Click here to connect to Apple's Gopher Server

The user double-clicks on a Mosaic page from within Gopher

The "Home Page" GopherItem knows that it needs to be displayed by a MosaicBrowser, so when it is opened it creates one and assigns the CyberItem of the MosaicBrowser's to be a copy of the "Home Page" GopherItem. The MosaicBr owser uses the GopherStream provided by the "Home Page" GopherItem to download the HTML page for display.



Data relationships resulting from opening a Mosaic document from within Gopher

Summary

In keeping with the OpenDoc philosophy, the Cyberdog architecture allows developers to write reusable components. The architecture addresses the additional challenge of obtaining the data from remote locations using different protocols by defining CyberItems and CyberStreams as unifying abstractions. With this model, developers can write components that interoperate smoothly. The examples above show that a single PictureViewer can display images regardless of their source, and that a single MosaicBrowser can display HTML documents r egardless of whether they reside on a Gopher or Mosaic server. This flexibility compares quite favorably with the two alternatives available today: 1) download the document and open it in another application, or 2) r eimplement the functionality of other applications in your own.

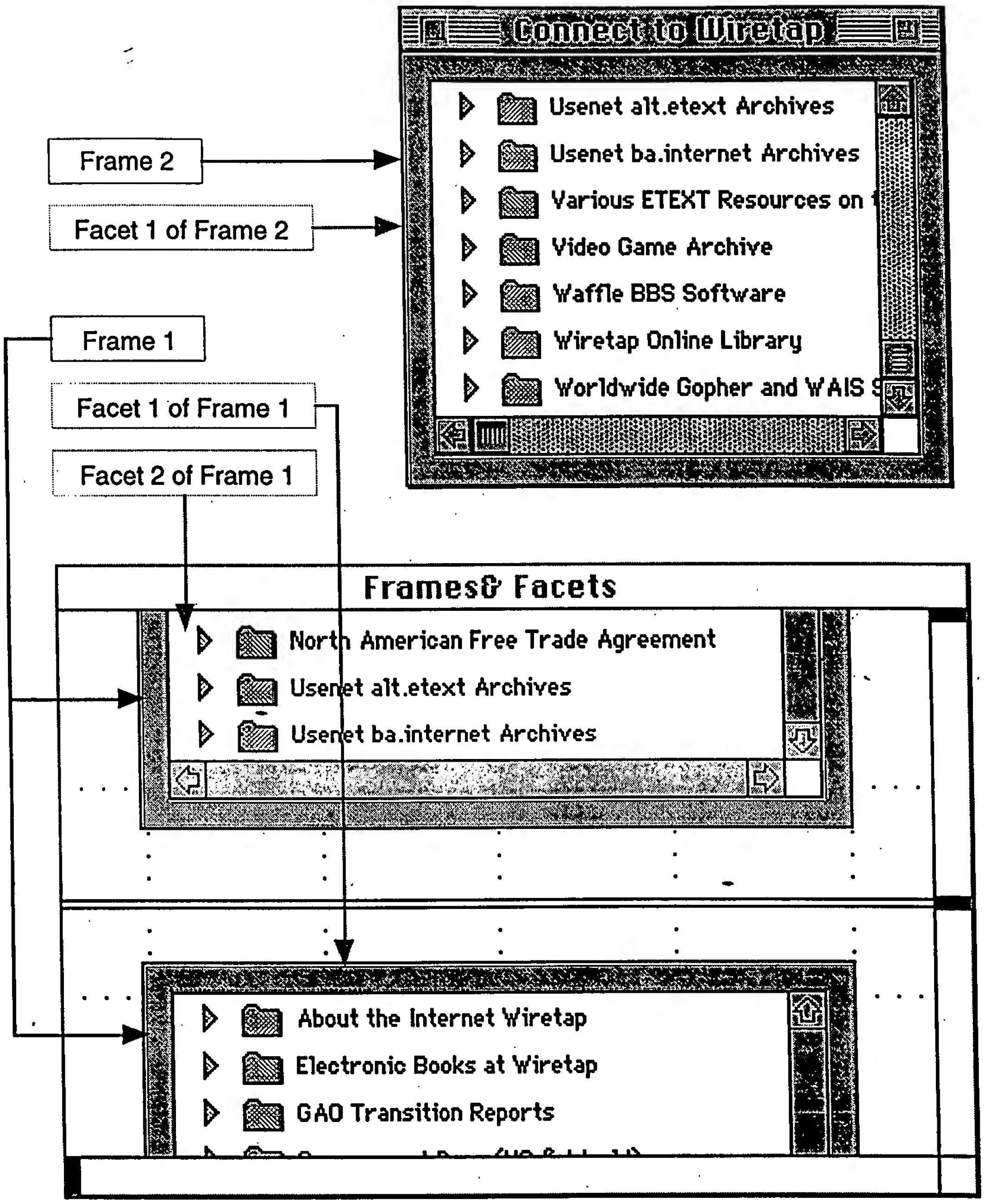
Internal Framework Implementation

OpenDoc Background: Frames and Facets

Since Cyberdog browsers and viewers are implemented as OpenDoc parts, they must support being embedded in other documents. Since the embedding part may support features such as split views, OpenDoc requires that all parts follow an imaging protocol. From an implementation perspective, the most challenging aspect is supporting frames and facets, which together define how a part appears on the screen. The following definitions are from the OpenDoc Class Reference:

Frames	An XMPFrame object (along with its facets) describes the geometric boundary between an embedded part and its containing part. A part may be displayed in multiple frames, and each frame represents a particular view of its parts content. Each frame may also appear in a window in multiple places, one for each facet of the frame.
Facets	Parts create facets for each place they display an embedded frame. In most cases, a containing part will only have one facet for an embedded frame in each facet of the containing part's display frame. But in some cases, a containing part will want to display the same embedded frame in several places. This can easily be accomplished by creating one facet for each place the containing part wants to display the embedded frame.

To make this more concrete, the following diagram shows the relationship between parts, frames, and facets. In this example, there is a Gopher part embedded in a drawing document. This is represented by a single frame, which has been split into two facets by the split bar owned by the draw part. Further, the user has used the "View as Window" command to hoist the Gopher browser into its own window. This is represented in OpenDoc by another frame. This frame displays the part in only one place and so has only a single facet. Thus there is a single part that is displayed in two frames and three facets.


Connect to Wiretap
Usenet alt.etext Archives
Usenet ba.internet Archives
Various ETEXT Resources on t
Video Game Archive
Waffle BBS Software
Wiretap Online Library
Worldwide Gopher and WAIS S
Frame 2
Facet 1 of Frame 2
Frame 1
Facet 1 of Frame 1
Facet 2 of Frame 1
Frames& Facets
North American Free Trade Agreement
Usenet alt.etext Archives
Usenet ba.internet Archives
About the Internet Wiretap
Electronic Books at Wiretap
GAO Transition Reports

A Gopher browser displayed in two frames with three facets

Goals

The Cyberdog view system was designed with these requirements in mind:

- Work with OpenDoc parts which may contain multiple frames: The same data may be displayed by views in multiple frames. Some views will have the same appearance in each frame (for example, spinners or buttons). Other views will have different appearances for the data (for example, a text view may wrap to a different rectangle).

- Work with OpenDoc parts which may be drawn in multiple facets: A view which is displayed in multiple facets of a frame must have the same appearance in each facet.
- The view system should support AppleScript.
- Views should work in both dialog boxes and OpenDoc parts.
- Views should work on multiple monitors, possibly with different bit depths.
- The view system should be extensible, able to add new view types without needing to update a central "create" function or bottleneck.
- The view system should allow for simple resizing. We assume that views resize or move relative to the boundary rectangle of all views, but views do not resize or move relative to neighboring views.
- As many Cyberdog windows will have a chiseled gray background, the view mechanism should do this automatically.

Implementation

Separation Between View Object and Data

The Cyberdog view system is like many other object-orientated view systems in that the view object is separated from the data that it displays. This separation between view and data allows a central authority (the data owner) to be the sole manipulator of the data and simplifies views in that they only have to know how to display the data. This also allows several view objects to display the same data at the same time.

As view objects are not allowed to directly change the data they display, a change mechanism (described below) is defined where a view object can request that the data owner change its data.

When the data owner changes the data (either because it received a change request from a view object or for some other reason — AppleEvents, asynchronous synching with reality, etc.) view objects are notified of the change via the change mechanism.

Change Mechanism

Information about changes to data is propagated via the TChange class. This class contains information that is common to all types of changes:

- the change event (i.e. data changed)
- a specifier for which view the change effects
- the distribution of the change (global to all frames or just the current one)[1]

Subclasses of the TChange class may add more information, such as the range of characters

[1] typically, global changes affect the data, where changes to the current frame affect the presentation

that have been deleted.

The information in a TChange class must be sufficient for the data owner to incorporate the change into the data. Additional information may be specified to make it easier for views to update their displays to reflect the change in the data.

TChanges flow in two directions: 1) view objects send TChanges to the data owner when they wish to change the data (such as when the user types a key on the keyboard); and 2) data owners propagate TChanges to views after they have changed the data.

Because all changes pass through a single bottleneck, TChanges can be easily recorded and manufactured, thereby supporting AppleScript recording and playback.

Views

Views are handled by TView objects. TViews contain information specific to the view that they are displaying:

- boundary rectangle
- font/size/style
- flags (visible, needs idle, resize flags [resize, reposition, both, or none])
- chisel options for creating three-dimensional borders (up, down, none)
- etc....

TView class

Examples of some Cyberdog views are:

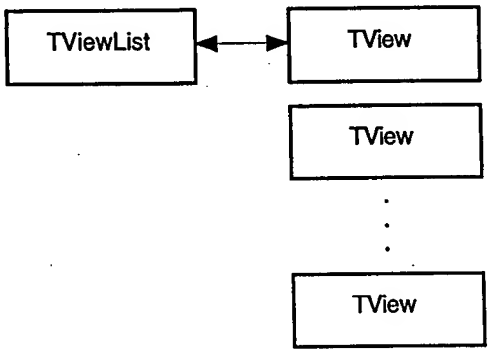
- Box
- Button
- Check box
- Tree - displays a hierarchical tree structure similar to the Finder's outline view
- Spinner - displays animated spinning arrows
- Text - displays rich text
- Picture
- Editable Text - displays simple editable text
- Static Text

The base TView handles:

- drawing a chiseled border, if necessary
- resizing the views boundary, when the view list (see below) boundary has changed
- maintaining the views flags

View Lists

Related views are grouped together via the TViewList class. TViewLists contain common information about the TViews: bounding rectangle of all views, background color, window the views are displayed in, etc., and a list of the TViews that are in the list:



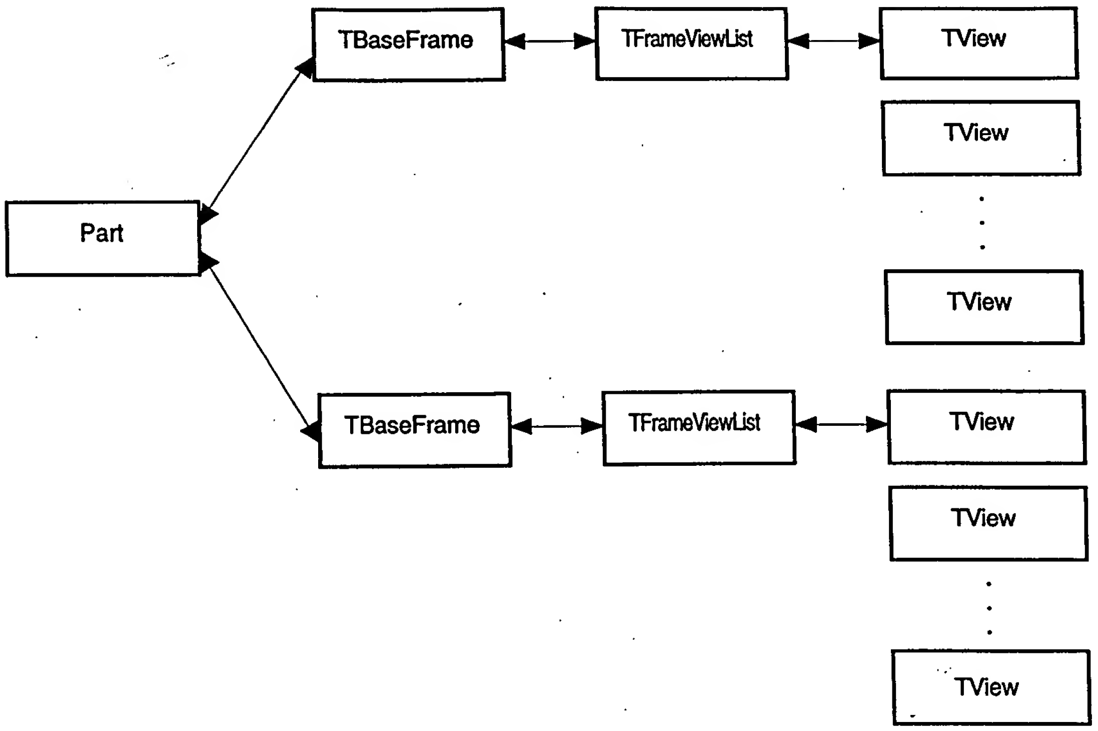
TViewList has a list of TView's

The TViewList class is responsible for:

- Conveying TChanges from the data owner to the TViews
- Conveying TChanges from TViews to the data owner
- Calling TViews when they need to be redrawn
- Drawing the background
- Resizing views when the boundary rectangle for the view list has changed
- Maintaining the "active" view (the one that gets keyboard events)
- Dispatching keyboard and mouse events to the TViews

Frame View Lists

For OpenDoc parts, TFrameViewList (a subclass of TViewList) is used. TFrameViewLists address communication of TChanges from views to the part (the data owner). Each OpenDoc part has a TBaseFrame object that handles frame-specific tasks. One of those tasks is to store and manipulate a TFrameViewList. For example, if a part had two frames, the following objects would be created:

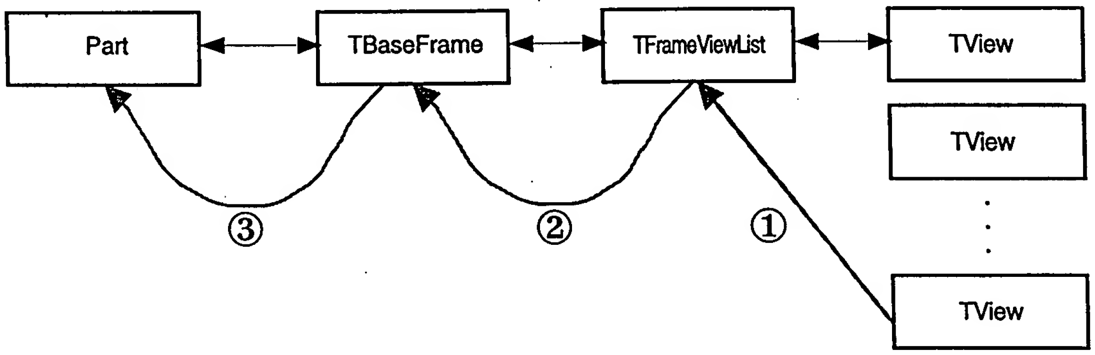


Objects created for a part with two frames

Change Propagation

Changes are propagated in two directions: *change requests* propagate from views up to data owners, *change notifications* propagate down from data owners to views.

Changes are propagated between TViews, TViewLists, TBaseFrames, and parts. The standard propagation of a change request starts with a TView:



Typical change request propagation

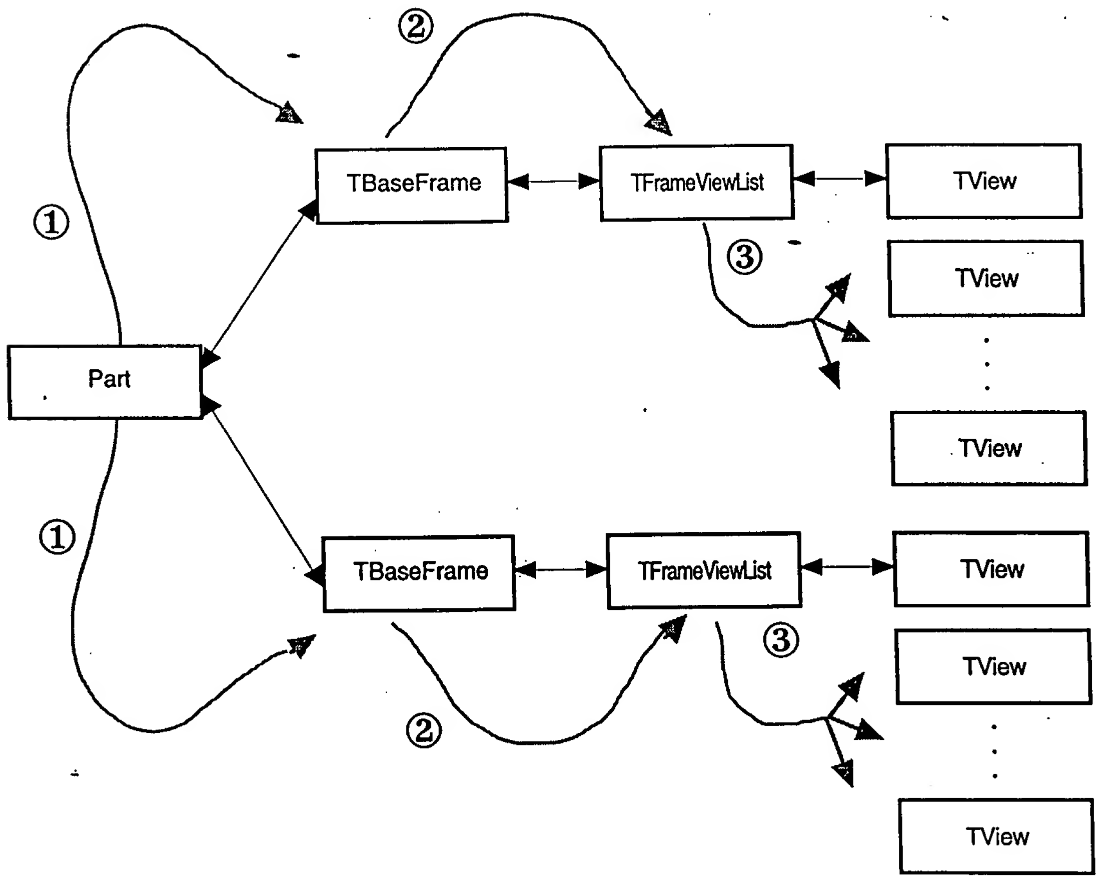
① A TView creates a TChange object (typically in response to a user action) and hands it off to it's TViewList (or TFrameViewList).

② The TViewList inspects the TChange and determines if it is the data owner for the change. If it is the data owner (such as when the view list is displayed in a dialog), the view list processes the change and propagates a change notification (see below) back to the TViews. If the view list is not the owner (such as when the view list is owned by a part), it hands the change to it's TBaseFrame.

③ The TBaseFrame is just like the TFrameViewList, processing the change if it belongs to the TBaseFrame (typically because the change is a appearance change) and passing it up to the Part if it is not (typically because the change is a data change).

While change requests are usually originated by views, change requests can be originated by any object in the change request propagation chain.

Change notifications get propagated in a similar manner. The standard propagation of a change notification starts with a part, either as a result of completing a change request or when the part asynchronously realizes its data has changed.



Typical change notification propagation

① After making a change to the data, the part manufactures a TChange (or it may reuse a change notification if it just completed the change) and sends it to each of its frames.

② The TBaseFrame notifies the view list of the change. If there are multiple facets for the frame, the view list is notified of the change once for each facet of the frame. This allows all facets to be updated.

③ The view list notifies views of the change.

While change notifications are usually originated by parts (or data owners), change notifications can be originated by any object in the change notifications propagation chain.

View Description and Instantiation

View lists are described by 'VIEW' resources. Each 'VIEW' resource contains global information about the view list (background color, bounding rectangle, etc.) and a list of views with their specific view information (options, bounding rectangle, etc.).

To instantiate a view list, the view list owner (usually a TBaseFrame in the case of OpenDoc parts) calls MakeViewList, specifying the resource ID for the 'VIEW' resource.

In order to provide an extensible view architecture, views are identified in the 'VIEW' resource by Class ID. This enables new views to be added in the future without having to change the base view list code (i.e. there is no bottleneck function that knows about all view types in advance).

Implementation: NetNews

Goals

Multiple news servers

Users will be allowed to access more than one news server. This will facilitate the creation of specialized news servers for local discussions, because users can access newsgroups on any number of servers.

Portability

Many potential users will use shared machines, such as those found in college computer clusters. Information on which articles a user has read will be stored in a portable document that can be used on any machine. This is a especially important for NetNews since it requires more state than other services.

Read each article only once

Articles are sometimes cross-posted to multiple newsgroups. When the user reads an article, Cyberdog will mark the article as having been read in all newsgroups in which the article was posted.

New information notification

Users will be automatically notified when new articles have arrived for any newsgroup that they have added to their Favorite Places book. Users will also be notified when new newsgroups are discovered.

Discussions

Cyberdog will display articles in a newsgroup sorted by topic so that users can read all related articles at once, without hunting for the others. We call each group of articles a "discussion." (They are also sometimes referred to as "threads.")

Rich content

We will allow the posting and reading of articles containing rich content, such as pictures, sounds, movies, and file enclosures.

Performance

To optimize performance, we will attempt to minimize communication with the news servers. Whenever possible we will cache information on the local machine. We will add more performance goals after further research by our Human Interface Maven.

Memory

We are attempting to conserve memory as much as possible to handle large numbers of articles in a group.

Implementation

Newsgroup/Article Databases

Cyberdog will build and maintain two databases. The first is a cache of all known newsgroups on all user-specified news servers. The second is a cache of the headers of the articles in each newsgroup. Once this information is cached locally on the user's machine, the server only needs to be accessed to retrieve the content of a specific article that the user opens. This will offer significant performance improvements because we will only access the server once, rather than many times.

Storage

The information on what articles have been read will be stored in the newsgroup CyberItem's storage unit. This will allow the user to move between machines without losing information on which articles they've already read. There are no restrictions on the number of newsgroups that can be added to a storage unit.

Rich Content

The Cyberdog news article viewer will support the MIME format. This will allow the user to create and view articles that contain picture, sounds, movies, and enclosed files.

News Engine

Cyberdog will contain a news engine (as part of it's CyberService) that will run continuously when the favorite places book is opened, or when any newsgroup part is opened. The engine will be responsible for creating and maintaining the newsgroup and article databases. The engine also services all requests for information by the other news components (browsers and viewers). This allows the engine to service requests using the databases or, if needed, obtain the information directly from the news server.

Newsgroup Browser

The newsgroup browser displays the list of newsgroups in a tree, much like the outline views in the Finder. Since the list of newsgroups obtained from the news server is actually a flat list, we organize our display by collecting newsgroup by the components in their names. If the user has specified more than one news server, the top level of the hierarchy will display the servers, and opening a server displays the top level of newsgroups available on that server.

The browser will use Cyberdog's Tree, TreeView, and TreeHandler classes to display and manipulate the newsgroup tree.

Article Browser

Double-clicking on a newsgroup icon opens the article browser. The browser divides the articles into discussions, and displays the list of discussions. Discussions that contain more than one article can be dinked open to display a list of articles that are in that discussion.

When an article is opened, the article browser creates a new window that behaves like a content viewer, displaying the contents of the article. This window also contains controls that allow the user to move between articles and between discussions without closing the window. This allows the user to read many articles without the overhead of closing and opening the window multiple times. Because of the need for tight communication between the content window and the browser window, all this functionality needs to be rolled into the article browser, rather than having a separate browser and viewer.

We do not create CyberItems for articles and discussions for the following reasons:

- They are extremely temporary and can disappear at any time
- There is no separate content viewer for articles
- Articles are not added to the book or log

Memory

In order to conserve memory, we will not create objects for each article in a newsgroup. Instead we will create several large blocks of memory to store information about all the articles in a group. This avoids having potentially hundreds of small blocks of data, as well as saving the overhead for each of those blocks.

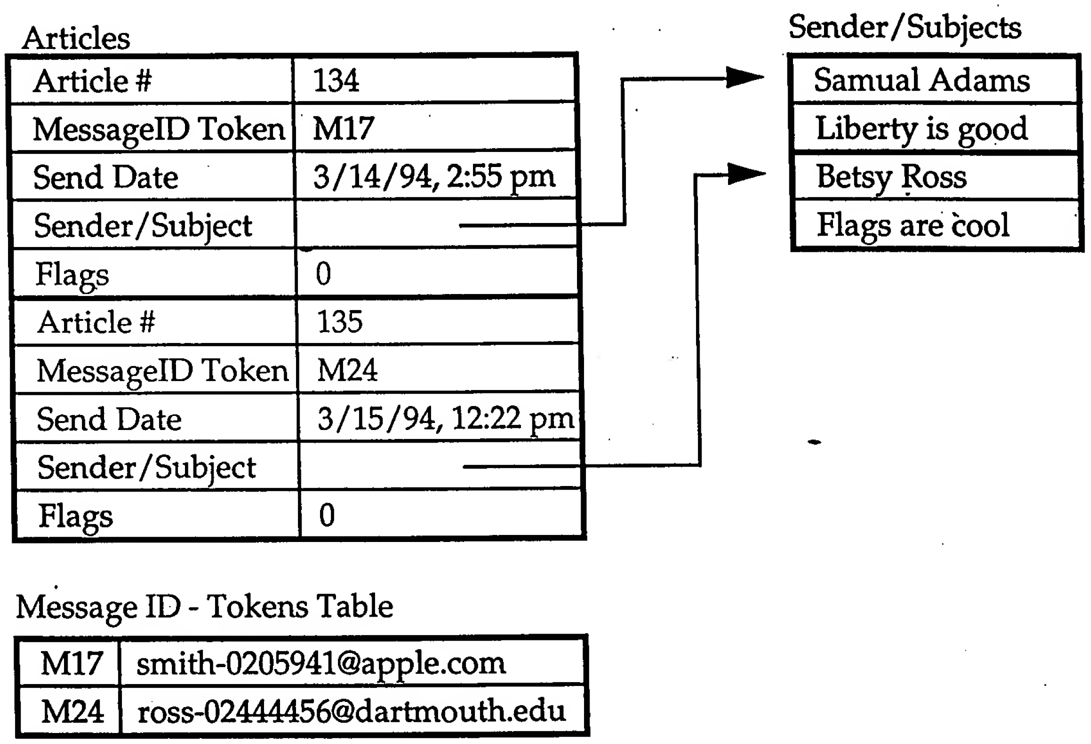
There are three main blocks of memory:

- *Sender/Subjects* - a list of string-pairs that consist of the sender's name and subject for each article. This list can be purged from memory and later reconstructed if the space is needed for other things.
- *Message ID Tokens* - a list of MessageID/Token pairs. This saves considerable memory because Message IDs are long strings, and we need to store them in several places. By "tokenizing" them, we can store a longint in those places. This mechanism will behave very similar to OpenDoc's token mechanism.
- *Articles* - a list of minimal required information about each article. For each article, we

store the following pieces of data:

- Article Number
- MessageID Token - allows us to retrieve the messageID
- Send Date
- Sender/Subject - a pointer into the sender/subject list
- Flags - attributes of the article, such as "has been read"

For our list of articles in memory, we only require 16 bytes/article. We also store a table of MessageID (string) to MessageID Token mappings. The messageIDs average around 32 characters in length. So this table requires 36 bytes/messageID. This means that for 400 articles, we use approximately (36 + 16) * 400 bytes, or 20k of memory.



Articles

Article #	134
MessageID Token	M17
Send Date	3/14/94, 2:55 pm
Sender/Subject	
Flags	0
Article #	135
MessageID Token	M24
Send Date	3/15/94, 12:22 pm
Sender/Subject	
Flags	0

Sender/Subjects

Samual Adams
Liberty is good
Betsy Ross
Flags are cool

Message ID - Tokens Table

M17	smith-0205941@apple.com
M24	ross-02444456@dartmouth.edu

The three main memory blocks

Displaying a newsgroup

Displaying the articles in the newsgroup is complicated because the user may want to divide the articles into "discussions" which are groups of related articles. So, before we can display any information in this manner, we need to group all the articles into discussions.

For each discussion, we keep two lists. The first is a list of all articles that we know are in this discussion. The second is a list of all message id tokens that we know are associated with this discussion. The second list is useful if some articles in the discussion have expired or if our server received the articles in the wrong order.

We obtain article headers from the news engine one at a time. For each article, we perform the following steps:

- Tokenize the messageID of this article, and the messageID of each article mentioned in the references section of the header. This may add some ID/Token pairs to the MessageID/Tokens table. Since each messageID is only tokenized once, we may not have to create any new tokens in this step.
- Add the article at the end of our (in memory) article list.
- Examine all discussions in the discussions list. If the article message token of the article (or the token for any of its references) is listed in any discussion's "reference" list, then add this article to that discussion. Otherwise create a new discussion and add this article to the new discussion. When an article is added to a discussion, the article is placed in the discussion's article list, and the article message token and the message token of all referenced articles are placed in the discussion's reference list.

Here is an example of breaking articles in a newsgroup into discussions. In this example, to make reading easier, we indicate a message id (typically a long string) by "MID-*n*", and message id tokens by "M*n*". The following is a list of our example articles:

Articles (from engine)

Article #:	MessageID:	References:
5	MID-1	<none>
6	MID-2	MID-7 MID-8
7	MID-3	MID-1
10	MID-4	MID-2
11	MID-5	MID-1 MID-3
12	MID-6	<none>
13	MID-7	<none>

These are the steps we go through to process the first article (Article-5):

- Tokenize MID-1 to M-1
- Add Article-5 to our (empty) article list
- Search our (empty) discussion list for any r eference to M-1. Since there is none, create a new discussion.
- Add M-1 to the discussion's reference list, and Article-5 to the article list.

Articles

Article#:	5
Message:	M1

Message IDs

MID-1	M1

Discussions

Articles:	5
References:	M1

Articles, MessageIDs, and Discussions after adding Article-5

Now we add the next article (Article-6)

- Tokenize MID-2 to M-2, MID-7 to M-7, and MID-8 to M-8.
- Add Article-6 to our article list
- Search our discussion list for any reference to M-2, M-7, or M-8. Since there are no references to these messages, create a new discussion.

- Add M-2, M-7, and M-8 to the discussion's reference list, and Article-6 to the article list.

Articles

Article#:	5
Message:	M1
Article#:	6
Message:	M2

Message IDs

MID-1	M1
MID-2	M2
MID-7	M7
MID-8	M8

Discussions

Articles:	5
References:	M1
Articles:	6
References:	M2, M7, M8

Articles, MessageIDs, and Discussions after adding Article-6

Now we add the next article (Article-7)

- Tokenize MID-3 to M-3.
- Add Article-7 to our article list.
- Search our discussion list for any reference to M-3 or M-1 (the referenced article). Since the first discussion references M-1, we add Article-7 to the first discussion's article list, and M-3 to the discussion's reference list.

Articles

Article#:	5
Message:	M1
Article#:	6
Message:	M2
Article#:	7
Message:	M3

Message IDs

MID-1	M1
MID-2	M2
MID-7	M7
MID-8	M8
MID-3	M3

Discussions

Articles:	5, 7
References:	M1, M3
Articles:	6
References:	M2, M7, M8

Articles, MessageIDs, and Discussions after adding Article-7

Here are what the lists look like after we add all the articles. Notice that we have three discussions. Discussion 1 has three articles (5, 7, and 11), Discussion 2 has three articles (6, 10, and 13) and Discussion 3 has one article (12).

Articles

Article#:	5
Message:	M1
Article#:	6
Message:	M2
Article#:	7
Message:	M3
Article#:	10
Message:	M4
Article#:	11
Message:	M5
Article#:	12
Message:	M6
Article#:	13
Message:	M7

MessageIDs

MID-1	M1
MID-2	M2
MID-3	M3
MID-4	M4
MID-5	M5
MID-6	M7
MID-7	M7
MID-8	M8

Discussions

Articles:	5, 7, 11
References:	M1, M3, M5
Articles:	6, 10, 13
References:	M1, M7, M8, M4
Articles:	12
References:	M6

Final Articles, MessageIDs, and Discussions

There is one case that the algorithm as described does not handle. In the following example article list, our news server received the articles out of order. The actual order of creation might be 5, 7, 6. Notice how the articles get divided into two discussions, rather than all appearing in the same discussion.

Articles

Article #:	5
MessageID:	MID-1
References:	<none>
Article #:	6
MessageID:	MID-2
References:	MID-3
Article #:	7
MessageID:	MID-3
References:	MID-1

Discussions

Articles:	5, 7
References:	M1, M3
Articles:	6
References:	M2, M3

Example requiring merging discussions

One possible solution to this problem is that as we process an article, we collect all discussions where we could insert the article, and merge them all into one discussion. In the above discussion, as we added Article-7, we would notice that it belongs in both the first and second discussion, so we would merge them both into a single discussion.

Implementation: Gopher Browser

The Gopher Browser was the first Cyberdog service to be implemented by the engineering team. It served as as a proof-of-concept vehicle for both the public and private Cyberdog frameworks.

Goals

Display

The Gopher Browser should display Gopher directories using Finder-like tree views. When a Gopher object is double-clicked, Cyberdog should be able to display that object in an appropriate viewer.

Simultaneous Asynchronous Enumerations

The user should be able to double-click on several items or dink several triangles to start simultaneous asynchronous enumerations. There should be clear feedback when enumerations are in process and the user should be able to safely cancel at any time.

Implementation

Display

There is very little actual engineering involved in writing the GopherBrowser class. Almost all of the interesting behavior is derived from the public Cyberdog classes or the internal Cyberdog framework. The most challenging aspect of the human interface, displaying a tree representing GopherSpace, is handled by Cyberdog's Tree, TreeView, and TreeHandler classes. The mechanism for opening the appropriate parts to display content is derived almost entirely from CyberItem and CyberPart. Making this mechanism work for Gopher simply meant putting a switch statement into TGopherItem::Open() to translate from standard Gopher types to OpenDoc part kinds.

Simultaneous Asynchronous Enumerations

The GopherBrowser class defines a TEnumeration class that associates a GopherStream with a particular node in a tree. (The GopherStream does all the work of downloading data from a Gopher Server.) The GopherBrowser maintains a list of outstanding enumerations, each of which gets processed at idle time. When new data arrives, the GopherBrowser updates its tree and then broadcasts a change to all of its TreeViewers to update the screen.

Canceling enumerations is handled in two ways:

1) By simply removing the last frame displaying a particular GopherBrowser. In practical terms, this is equivalent to closing the window for virtually all cases.

2) The user can also cancel an enumeration by "up-dinking" a node's triangle. This will cancel all enumerations associated with any descendants of the node.

Implementation: Log

Goals

Log all Transitions

The log is intended to record all transitions a user makes from one "place" on the Internet to another. The exact notion of a place will vary from service to service, but any service that allows browsing should make use of the log. It is especially important that third party services be able to use the log in the same manner as those shipped with the product.

Heterogeneous Data

Because of the above requirement, the log must accept data representing a variety of different types: newsgroups, gopher directories, mosaic pages,and Internet "places" that are not yet invented.

Works in Non-Book Environments

Because Cyberdog services are implemented as OpenDoc parts, they can be embedded in any type of document. A CyberPart cannot assume that it was launched from within the favorite places book, so the log should work regardless of where a service originated. For example, if a user begins exploring the Internet from a Gopher part embedded in a word processing document, the log should still record their path.

Implementation

Log all Transitions

To make the log available to all services, the AddCyberItemToLog method has been added to TCyberPart. All objects that derive from TCyberPart (which will include all browsers and viewers developed by both the Cyberdog team and third parties) can determine when a transition from one place to another has occurred and call AddCyberItemToLog.

Heterogeneous Data

The log uses CyberItems to unify the various data types that can be added to the log. To add an item, A CyberPart calls AddCyberItemToLog with two parameters: one CyberItem representing the parent "place" and one representing the child. The log then adds a node to the tree in the appropriate place by comparing CyberItems. To make this possible, the log maintains an ordered list of all CyberItems and their corresponding nodes in the tree. (Note: if performance tests on large logs warrant it, the ordered list may be replaced with a hash table or an AVL tr ee.)

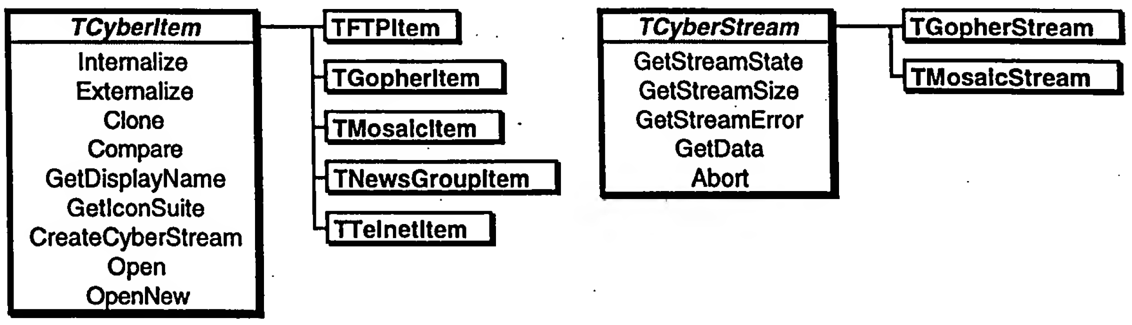
CyberItems are ordered by an arbitrary (but consistent) ordering among subclasses of CyberItem, and each subclass is then responsible for determining an ordering among its members.

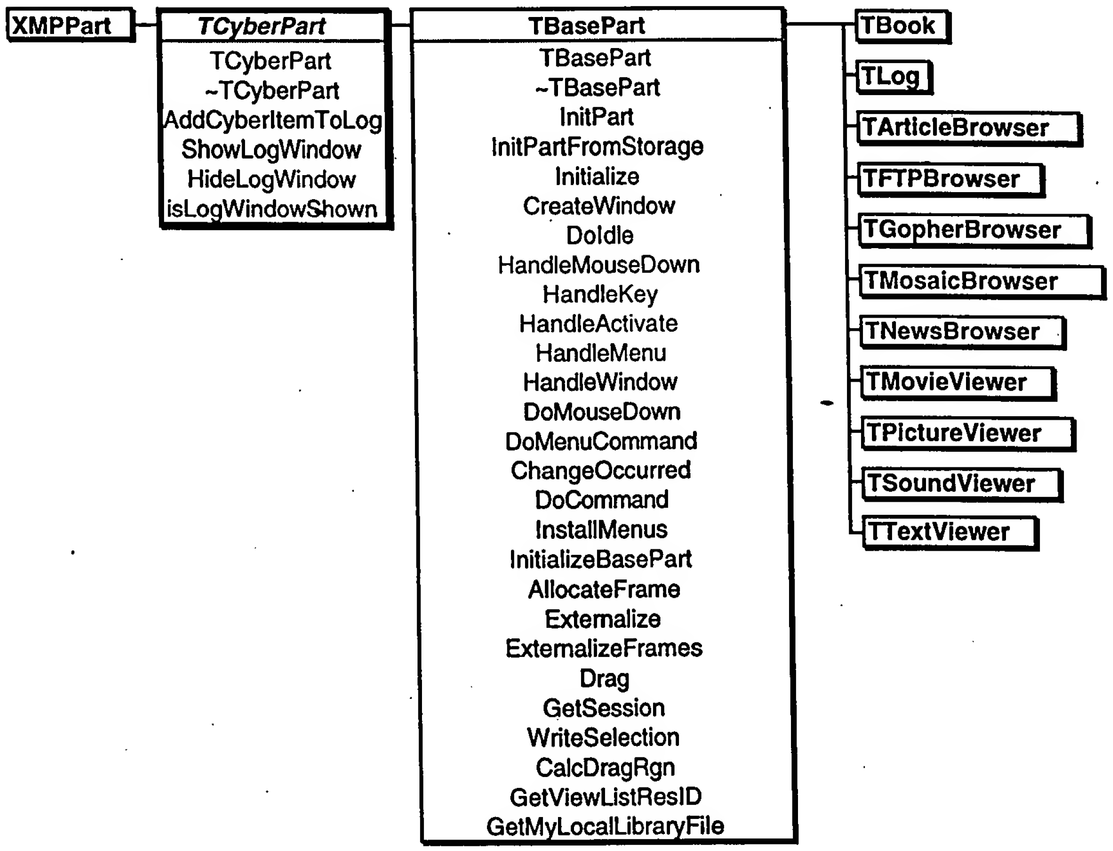
Works in Non-Book Environments

Whenever TCyberPart::AddCyberItemToLog is called, it first calls the static (class) method TLog::ObtainLog to make sure that there is a log associated with the current OpenDoc session. (A session corresponds to a process.) If a log does not exist, TLog::ObtainLog creates one and uses the OpenDoc name space mechanism to store a pointer to it in a place that can be accessed from any part in the same session.

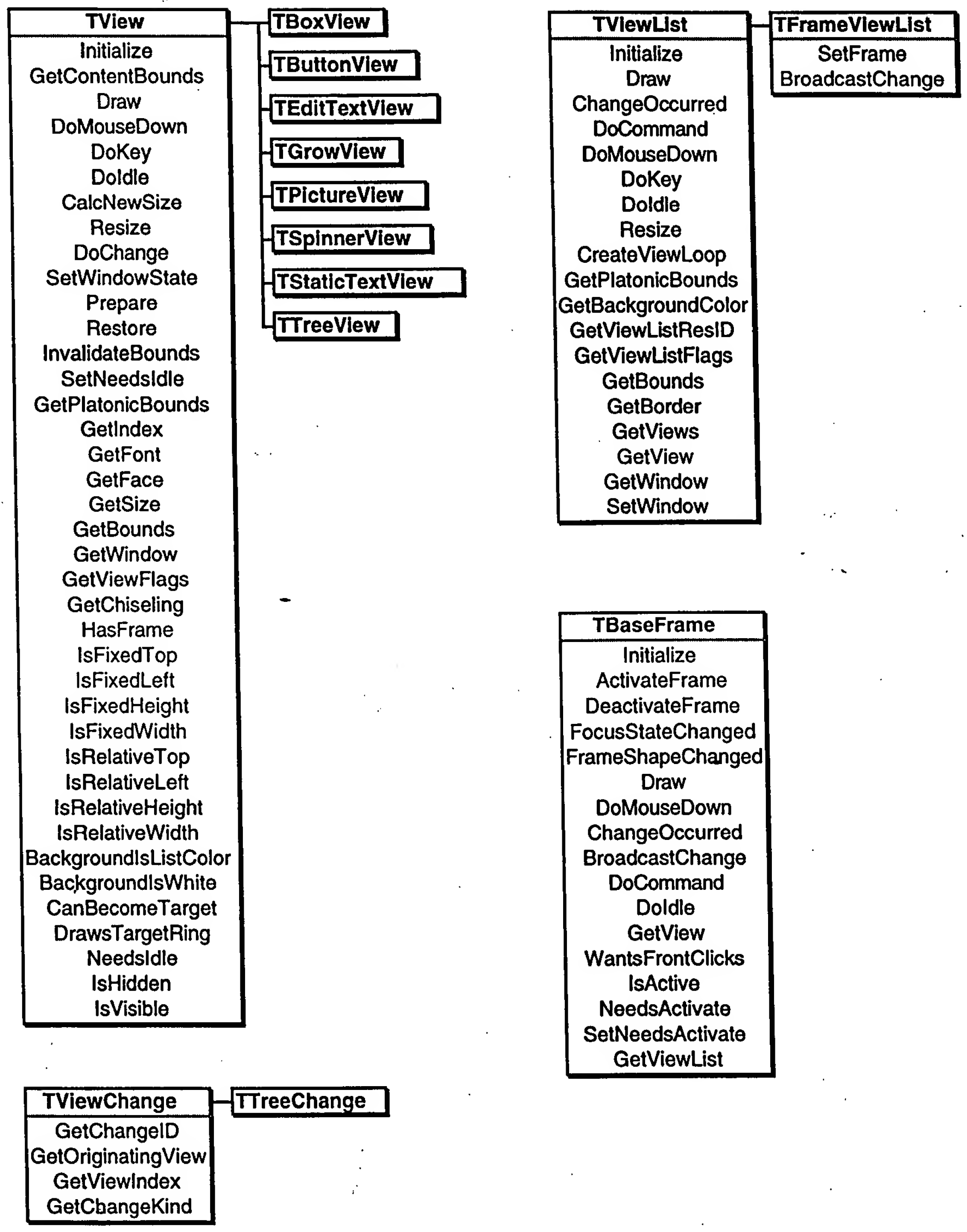
Cyberdog Class Hierarchy

Cyber Classes and Descendants

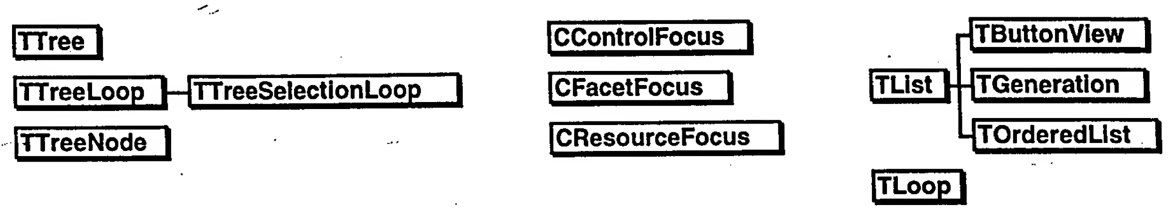




View Mechanism



Utility Classes


TTree
TTreeLoop
TTreeSelectionLoop
TTreeNode
CControlFocus
CFacetFocus
CResourceFocus
TList
TButtonView
TGeneration
TOrderedList
TLoop